目录

Nginx学习笔记

Nginx是一个高性能的HTTP和反向代理服务器，以高稳定性、丰富的功能集、示例配置文件和低系统资源的消耗而闻名。

这是个人学习Nginx的笔记，请点击下面的链接阅读或者下载电子版本:

- 在线阅读
  - 国外服务器：gitbook提供的托管，服务器在国外，速度比较慢，经常被墙
  - 国内服务器：腾讯云加速，国内网速极快
- 下载pdf格式
- 下载mobi格式
- 下载epub格式

Nginx 介绍

安装

ubuntu 16.04

在 ubuntu 的默认仓库中就有 nginx，因此安装非常简单，直接 apt-get 命令即可：

```
sudo apt-get install nginx
```

参考资料

- How To Install Nginx on Ubuntu 16.04

配置

虚拟主机

tags:虚拟主机

介绍

虚拟主机使用的是特殊的软硬件技术，把一台运行在因特网上的服务器主机分成多台"虚拟"的主机，每台虚拟主机是一个独立的网站，具有独立的域名，具有完整的Intemet服务器功能（WWW、FTP、Email等）。同一台主机上的虚拟主机之间是完全独立的，在网站访问者看来，每一台虚拟主机和一台独立的主机是完全一样。

总结：虚拟主机提供了在同一台服务器、同一个 Nginx 进程上运行多个网站的功能。

支持

Nginx可以配置多种类型的虚拟主机。

基于IP的虚拟主机

Linux操作系统允许添加IP别名。

> 注：IP别名指在一块物理网卡上绑定多个IP地址。

IP别名能够在使用单一网卡的同一个服务器上运行多个基于IP的虚拟主机。

操作系统设置IP别名非常容易，只须配置系统上的网络接口，让它监听额外的IP地址。在Linux系统上，可以使用标准的网络配置工具（比如ifconfig和route命令）添加IP别名。

基于域名的虚拟主机

基于端口的虚拟主机

实战：基于域名的虚拟主机

tags:实战,虚拟主机

前言

默认包含

默认在 `/etc/nginx/nginx.conf` 配置文件中会有如下配置：

```
http {
    ......
    ##
    # Virtual Host Configs
    ##

    include /etc/nginx/conf.d/*.conf;
    include /etc/nginx/sites-enabled/*;
}
```

这表明默认情况下 nginx 会自动包含 `/etc/nginx/conf.d/*.conf` 和 `/etc/nginx/sites-enabled/*` 。

启用站点和可用站点

默认情况下，在 `/etc/nginx/sites-enabled` 下有一个默认站点，这个站点也就是 nginx 安装之后的默认站点：

```
$ cd /etc/nginx/sites-enabled
$ ls -l
total 0
lrwxrwxrwx 1 root root 34 Oct  6 02:19 default -> /etc/nginx/sites-available/default
```

打开 `/etc/nginx/sites-available/default` 可以看到如下内容：

```
server {
        listen 80 default_server;
        listen [::]:80 default_server;

        root /var/www/html;
        index index.html index.htm index.nginx-debian.html;

        server_name _;
        location / {
                try_files $uri $uri/ =404;
        }
```

按照这个文档的建议，最好是在 `/etc/nginx/sites-available/` 下建立站点的配置文件，这些站点就是所谓的"可用站点"。然后在 link 到 `/etc/nginx/sites-enabled` 下开启站点，这些开启的站点就是所谓"启用站点"。

通过建立链接来控制可用站点的启用。

实战

创建虚拟主机 **basiccloud.net**

目标： http://basiccloud.net 和 http://www.basiccloud.net 应该都指向同一个虚拟主机。

在 `/etc/nginx/sites-available/` 下新建 basiccloud.net 文件，内容如下：

```
server {
       listen 80;

       server_name basiccloud.net www.basiccloud.net;

       root /var/www/basiccloud.net;
       index index.html;
}
```

然后建立 `/var/www/basiccloud.net` 目录，准备好站点的html文件。

将 basiccloud.net 站点文件链接到 `/etc/nginx/sites-enabled/` 目录：

```
sudo ln -s /etc/nginx/sites-available/basiccloud.net /etc/nginx/sites-enabled/basicclo
ud.net
```

配置完成之后，在重新转载前，先验证一下：

```
$ sudo nginx -t
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
```

验证通过，再重新装载：

```
sudo nginx -s reload
```

创建虚拟主机 dolphin.basiccloud.net

目标： http://dolphin.basiccloud.net 应该指向另外一个虚拟主机。

在 `/etc/nginx/sites-available/` 下新建 dolphin.basiccloud.net 文件，内容如下：

```
server {
       listen 80;

       server_name dolphin.basiccloud.net;

       root /var/www/dolphin.basiccloud.net;
       index index.html;
}
```

然后建立 `/var/www/dolphin.basiccloud.net` 目录，准备好站点的html文件。

将 dolphin.basiccloud.net 站点文件链接到 `/etc/nginx/sites-enabled/` 目录：

```
sudo ln -s /etc/nginx/sites-available/dolphin.basiccloud.net /etc/nginx/sites-enabled/
dolphin.basiccloud.net
```

反向代理

实战：域名免端口

tags:实战,反向代理

前言

在实际使用中，由于web服务器启动于不同进程，因此需要指定不同的端口，也就意味着必然有web应用要使用80之外的端口，这样在地址栏中就必须出现端口号，非常影响用户体验。

比较好的方式，通过使用不同的域名或者二级域名，然后通过nginx反向代理的方式转发请求给到实际负责处理的服务器。

下面是这种方式的典型使用场景的例子：

实际**web**服务器的地址	反向代理之后给到终端用户的地址
*：8800	http://git.basiccloud.net
*：8081	http://maven.basiccloud.net

实战

创建虚拟主机 **git.basiccloud.net**

目标：http://git.basiccloud.net 应该指向当前机器上运行于 8800 端口的 gitlab 服务器。

在 `/etc/nginx/sites-available/` 下新建 git.basiccloud.net 文件，内容如下：

```
server {
        listen 80;

        server_name git.basiccloud.net;

        location /
        {
                proxy_pass http://127.0.0.1:8800;
        }
}
```

将 git.basiccloud.net 站点文件链接到 `/etc/nginx/sites-enabled/` 目录：

```
sudo ln -s /etc/nginx/sites-available/git.basiccloud.net /etc/nginx/sites-enabled/git.
basiccloud.net
```

修改完成之后，使用命令检测配置修改结果并重新装载配置：

```
sudo nginx -t
sudo nginx -s reload
```

创建虚拟主机 maven.basiccloud.net

目标：http://maven.basiccloud.net 应该指向当前机器上运行于 8081 端口的 artifactory 服务器。

在 `/etc/nginx/sites-available/` 下新建 maven.basiccloud.net 文件，内容如下：

```
server {
       listen 80;

       server_name maven.basiccloud.net;

       location /
       {
              proxy_pass http://127.0.0.1:8081;
       }
}
```

将 maven.basiccloud.net 站点文件链接到 `/etc/nginx/sites-enabled/` 目录：

```
sudo ln -s /etc/nginx/sites-available/maven.basiccloud.net /etc/nginx/sites-enabled/ma
ven.basiccloud.net
```

支持keep alive长连接

当使用nginx作为反向代理时，为了支持长连接，需要做到两点：

1. 从client到nginx的连接是长连接
2. 从nginx到server的连接是长连接

从HTTP协议的角度看，nginx在这个过程中，对于客户端它扮演着HTTP服务器端的角色。而对于真正的服务器端（在nginx的术语中称为upstream）nginx又扮演着HTTP客户端的角色。

保持和client的长连接

为了在client和nginx之间保持上连接，有两个要求：

1. client发送的HTTP请求要求keep alive
2. nginx设置上支持keep alive

HTTP配置

默认情况下，nginx已经自动开启了对client连接的keep alive支持。一般场景可以直接使用，但是对于一些比较特殊的场景，还是有必要调整个别参数。

需要修改nginx的配置文件(在nginx安装目录下的conf/nginx.conf):

```
http {
    keepalive_timeout  120s 120s;
    keepalive_requests 10000;
}
```

keepalive_timeout指令

keepalive_timeout指令的语法：

```
Syntax:    keepalive_timeout timeout [header_timeout];
Default:    keepalive_timeout 75s;
Context:    http, server, location
```

第一个参数设置keep-alive客户端连接在服务器端保持开启的超时值。值为0会禁用keep-alive客户端连接。可选的第二个参数在响应的header域中设置一个值“Keep-Alive: timeout=time”。这两个参数可以不一样。

> 注：默认75s一般情况下也够用，对于一些请求比较大的内部服务器通讯的场景，适当加大为120s或者300s。第二个参数通常可以不用设置。

keepalive_requests指令

keepalive_requests指令用于设置一个keep-alive连接上可以服务的请求的最大数量。当最大请求数量达到时，连接被关闭。默认是100。

> 这个参数的眞实含义，是指一个keep alive建立之后，nginx就会为这个连接设置一个计数器，记录这个keep alive的长连接上已经接收并处理的客户端请求的数量。如果达到这个参数设置的最大值时，则nginx会强行关闭这个长连接，逼迫客户端不得不重新建立新的长连接。
>
> 这个参数往往被大多数人忽略，因为大多数情况下当QPS(每秒请求数)不是很高时，默认值100凑合够用。但是，对于一些QPS比较高（比如超过10000QPS，甚至达到30000,50000甚至更高) 的场景，默认的100就显得太低。
>
> 简单计算一下，QPS=10000时，客户端每秒发送10000个请求(通常建立有多个长连接)，每个连接只能最多跑100次请求，意味着平均每秒钟就会有100个长连接因此被nginx关闭。同样意味着为了保持QPS，客户端不得不每秒中重新新建100个连接。因此，如果用netstat命令看客户端机器，就会发现有大量的TIME_WAIT的socket连接(即使此时keep alive已经在client和nginx之间生效)。
>
> 因此对于QPS较高的场景，非常有必要加大这个参数，以避免出现大量连接被生成再抛弃的情况，减少TIME_WAIT。

保持和server的长连接

为了让nginx和server（nginx称为upstream）之间保持长连接，典型设置如下：

```
http {
    upstream  BACKEND {
        server   192.168.0.1：8080  weight=1 max_fails=2 fail_timeout=30s;
        server   192.168.0.2：8080  weight=1 max_fails=2 fail_timeout=30s;

        keepalive 300;        // 这个很重要！
    }

    server {
        listen 8080 default_server;
        server_name "";

        location /  {
            proxy_pass http://BACKEND;
            proxy_set_header Host  $Host;
            proxy_set_header x-forwarded-for $remote_addr;
            proxy_set_header X-Real-IP $remote_addr;
            add_header Cache-Control no-store;
            add_header Pragma  no-cache;

            proxy_http_version 1.1;                    // 这两个最好也设置
            proxy_set_header Connection "";

            client_max_body_size  3072k;
            client_body_buffer_size 128k;
        }
    }
}
```

upstream设置

upstream设置中，有个参数要特别的小心，就是这个keepalive。

大多数未仔细研读过nginx的同学通常都会误解这个参数，有些人理解为这里的keepalive是设置是否打开长连接，以为应该设置为on/off。有些人会被前面的keepalive_timeout误导，以为这里也是设置keepalive的timeout。

但是实际上这个keepalive参数的含义非常的奇特，请小心看nginx文档中的说明:

```
Syntax:    keepalive connections;
Default:    —
Context:    upstream
```

> Activates the cache for connections to upstream servers. 激活到upstream服务器的连接缓存。
>
> The connections parameter sets the maximum number of idle keepalive connections to upstream servers that are preserved in the cache of each worker process. When this number is exceeded, the least recently used connections are closed. connections参数设置每个worker进程在缓冲中保持的到upstream服务器的空闲keepalive连接的最大数量. 当这个数量被突破时，最近使用最少的连接将被关闭。
>
> It should be particularly noted that the keepalive directive does not limit the total number of connections to upstream servers that an nginx worker process can open. The connections parameter should be set to a number small enough to let upstream servers process new incoming connections as well. 特别提醒：keepalive指令不会限制一个nginx worker进程到upstream服务器连接的总数量。connections参数应该设置为一个足够小的数字来让upstream服务器来处理新进来的连接。

在这里可以看到，前面的几种猜测可以确认是错误的了：

1. keepalive不是on/off之类的开关
2. keepalive不是timeout，不是用来设置超时值

很多人读到这里的文档之后，会产生另外一个误解：认为这个参数是设置到upstream服务器的长连接的数量，分歧在于是最大连接数还是最小连接数，不得不说这也是一个挺逗的分歧.....

回到nginx的文档，请特别注意这句话，至关重要：

> The connections parameter sets the maximum number of idle keepalive connections to upstream servers connections参数设置到upstream服务器的空闲keepalive连接的最大数量

请仔细体会这个"idle"的概念，何为idle。大多数人之所以误解为是到upstream服务器的最大长连接数，一般都是因为看到了文档中的这句话，而漏看了这个"idle"一词。

然后继续看文档后面另外一句话：

> When this number is exceeded, the least recently used connections are closed. 当这个数量被突破时，最近使用最少的连接将被关闭。

这句话更是大大强化了前面关于keepalive设置的是最大长连接数的误解：如果连接数超过keepalive的限制，就关闭连接。这不是赤裸裸的最大连接数么？

但是nginx的文档立马给出了指示，否定了最大连接数的可能：

> It should be particularly noted that the keepalive directive does not limit the total number of connections to upstream servers that an nginx worker process can open. 特别提醒：keepalive指令不会限制一个nginx worker进程到upstream服务器连接的总数量。

keepalive参数的理解

要真正理解keepalive参数的含义，请回到文档中的这句：

> The connections parameter sets the maximum number of idle keepalive connections to upstream servers connections参数设置到upstream服务器的空闲keepalive连接的最大数量

请注意空闲keepalive连接的最大数量中空闲这个关键字。

为了能让大家理解这个概念，我们先假设一个场景： 有一个HTTP服务，作为upstream服务器接收请求，响应时间为100毫秒。如果要达到10000 QPS的性能，就需要在nginx和upstream服务器之间建立大约1000条HTTP连接。nginx为此建立连接池，然后请求过来时为每个请求分配一个连接，请求结束时回收连接放入连接池中，连接的状态也就更改为idle。

我们再假设这个upstream服务器的keepalive参数设置比较小，比如常见的10.

假设请求和响应是均匀而平稳的，那么这1000条连接应该都是一放回连接池就立即被后续请求申请使用，线程池中的idle线程会非常的少，趋进于零。我们以10毫秒为一个单位，来看连接的情况(注意场景是1000个线程+100毫秒响应时间，每秒有10000个请求完成)：

- 每10毫秒有100个新请求，需要100个连接
- 每10毫秒有100个请求结束，可以释放100个连接
- 如果请求和应答都均匀，则10毫秒内释放的连接刚好够用，不需要新建连接，连接池空闲连接为零

然后再回到现实世界，请求通常不是足够的均匀和平稳，为了简化问题，我们假设应答始终都是平稳的，只是请求不平稳，第一个10毫秒只有50,第二个10毫秒有150：

1. 下一个10毫秒，有100个连接结束请求回收连接到连接池，但是假设此时请求不均匀10毫秒内没有预计的100个请求进来，而是只有50个请求。注意此时连接池回收了100个连接又分配出去50个连接，因此连接池内有50个空闲连接。
2. 然后注意看keepalive=10的设置，这意味着连接池中最多容许保留有10个空闲连接。因此nginx不得不将这50个空闲连接中的40个关闭，只留下10个。
3. 再下一个10个毫秒，有150个请求进来，有100个请求结束任务释放连接。150 - 100 = 50,空缺了50个连接，减掉前面连接池保留的10个空闲连接，nginx不得不新建40个新连接来满足要求。

我们可以看到，在短短的20毫秒内，仅仅因为请求不够均匀，就导致nginx在前10毫秒判断空闲连接过多关闭了40个连接，而后10毫秒又不得不新建40个连接来弥补连接的不足。

再来一次类似的场景，假设请求是均匀的，而应答不再均匀，前10毫秒只有50个请求结束，后10毫秒有150个：

1. 前10毫秒，进来100个请求，结束50个请求，导致连接不够用，nginx为此新建50个连接
2. 后10毫秒，进来100个请求，结束150个请求，导致空闲连接过多，ngixn为此关闭了150-100-10=40个空闲连接

特别提醒的时，第二个应答不均匀的场景实际上是对应第一个请求不均匀的场景：正是因为请求不均匀，所以导致100毫秒之后这些请求的应答必然不均匀。

现实世界中的请求往往和理想状态有巨大差异，请求不均匀，服务器处理请求的时间也不平稳，这理论上的大概1000个连接在反复的回收和再分配的过程中，必然出现两种非常矛盾场景在短时间内反复： 1. 连接不够用，造成新建连接 2. 连接空闲，造成关闭连接。从而使得总连接数出现反复震荡，不断的创建新连接和关闭连接，使得长连接的效果被大大削弱。

造成连接数量反复震荡的一个推手，就是这个keepalive 这个最大空闲连接数。毕竟连接池中的1000个连接在频繁利用时，出现短时间内多余10个空闲连接的概率实在太高。因此为了避免出现上面的连接震荡，必须考虑加大这个参数，比如上面的场景如果将keepalive设置为100或者200,就可以非常有效的缓冲请求和应答不均匀。

总结

keepalive 这个参数一定要小心设置，尤其对于QPS比较高的场景，推荐先做一下估算，根据QPS和平均响应时间大体能计算出需要的长连接的数量。比如前面10000 QPS和100毫秒响应时间就可以推算出需要的长连接数量大概是1000. 然后将keepalive设置为这个长连接数量的10%到30%。

比较懒的同学，可以直接设置为keepalive=1000之类的，一般都OK的了。

location设置

location中有两个参数需要设置：

```
http {
    server {
        location /  {
            proxy_http_version 1.1;                    // 这两个最好也设置
            proxy_set_header Connection "";
        }
    }
}
```

HTTP协议中对长连接的支持是从1.1版本之后才有的，因此最好通过proxy_http_version指令设置为"1.1"，而"Connection" header应该被清理。清理的意思，我的理解，是清理从client过来的http header，因为即使是client和nginx之间是短连接，nginx和upstream之间也是可以开启长连接的。这种情况下必须清理来自client请求中的"Connection" header。

Nginx官方文档

nginx documentation

这是nginx官方网站的文档。

后面选择性的翻译了其中比较关注的部分章节，对于负载均衡而言，以下两个章节是必须的：

- 使用nginx实现HTTP负载均衡
- Upstream设置文档

为了更好的理解nginx是如何工作，以及server_name的配置，这些内容可以作为预备知识：

- nginx如何处理请求
- 服务器名称

nginx如何处理请求

> 注：内容翻译自Nginx官网文档 How nginx processes a request。

基于名称的虚拟服务器

nginx首先要决定哪个服务器应该处理请求。让我们从一个简单的配置开始，三个虚拟服务器都监听在端口*:80:

```
server {
    listen      80;
    server_name example.org www.example.org;
    ...
}

server {
    listen      80;
    server_name example.net www.example.net;
    ...
}

server {
    listen      80;
    server_name example.com www.example.com;
    ...
}
```

在这个配置中，nginx仅仅检验请求header中的"Host"域来决定请求应该路由到哪个服务器。如果它的值不能匹配任何服务器，或者请求完全没有包含这个header域，那么nginx将把这个请求路由到这个端口的默认服务器。在上面的配置中，默认服务器是第一个 - 这是nginx标准的默认行为。也可以通过listen指令的default_server属性来显式的设置默认服务器:

```
server {
    listen      80 default_server;
    server_name example.net www.example.net;
    ...
}
```

> default_server 参数从版本0.8.21开始可用，在更早的版本中要使用default参数。

注意默认服务器是监听端口的一个属性，而不是服务器名称。后面会有更多描述。

如何防止使用未定义的服务器名称来处理请求

如果容许请求没有"Host" header 域，放弃这些请求的服务器可以定义为：

```
server {
    listen      80;
    server_name "";
    return      444;
}
```

这里，服务器名称被设置为空字符串，这样将匹配没有"Host"header域的请求， 并返回一个特殊的nginx的非标准码404，然后关闭连接。

> 从版本0.8.48开始，这是服务器名称的默认设置， 因此server_name ""可以不用写。在更早的版本中，机器的hostname被用作默认服务器名称。

基于名称和基于IP混合的虚拟服务器

让我们看一下更复杂的配置，有一些虚拟服务器监听在不同的地址：

```
server {
    listen      192.168.1.1:80;
    server_name example.org www.example.org;
    ...
}

server {
    listen      192.168.1.1:80;
    server_name example.net www.example.net;
    ...
}

server {
    listen      192.168.1.2:80;
    server_name example.com www.example.com;
    ...
}
```

在这个配置中，nginx首先通过server块的listen指令检验请求的IP地址和端口。然后在通过server块的server_name入口检验请求的"Host"header域。如果服务器名称没有找到，请求将被默认服务器处理。例如，在端口192.168.1.1:80接收到的去www.example.com的请求将被端口192.168.1.1:80的默认服务器处理。这里是第一个服务器，因为这个端口没有定义www.example.com。

前面已经提到，默认服务器是监听端口的属性，并且不同的端口可以定义不同的默认服务器：

```
server {
    listen      192.168.1.1:80;
    server_name example.org www.example.org;
    ...
}

server {
    listen      192.168.1.1:80 default_server;
    server_name example.net www.example.net;
    ...
}

server {
    listen      192.168.1.2:80 default_server;
    server_name example.com www.example.com;
    ...
}
```

一个简单的PHP站点配置

现在让我们看一下nginx如何选择location来为典型而简单的PHP站点处理请求：

```
server {
    listen      80;
    server_name example.org www.example.org;
    root        /data/www;

    location / {
        index   index.html index.php;
    }

    location ~* \.(gif|jpg|png)$ {
        expires 30d;
    }

    location ~ \.php$ {
        fastcgi_pass  localhost:9000;
        fastcgi_param SCRIPT_FILENAME
                      $document_root$fastcgi_script_name;
        include       fastcgi_params;
    }
}
```

nginx首先搜索由书面字符串给定的最为特别的前缀location，无视列表顺序。在上面的配置中仅有一个带前缀的location "/"，并且因为它匹配任何请求，它将用于作为最后的对策。然后nginx检查通过正则表达式给定的location，基于在配置文件中列出的顺序。第一个匹配的表达式将停止搜索然后nginx将使用这个location。如果没有正则表达式匹配请求，则nginx将使用前面发现的最为特别的前缀location。

注意所有类型的location仅仅检验请求行(HTTP中的request line)中的URL部分，不带参数。这是因为请求字符串中的参数可以以多种方式给出，例如：

```
/index.php?user=john&page=1
/index.php?page=1&user=john
```

还有，任何人都可能使用这样的查询字符串来请求：

```
/index.php?page=1&something+else&user=john
```

现在让我们来看在上面的配置中请求将如何被处理：

- 请求“/logo.gif” 首先匹配前缀location “/” 然后匹配正则表达式“.(gif|jpg|png)$”, 因此, 它被后面的location处理。使用指令“root /data/www”，请求被映射到文件/data/www/logo.gif, 然后文件被发送到客户端。
- 请求 “/index.php” 同样首先匹配到前缀location “/” 然后匹配正则表达式“.(php)$”. 因此, 它被后面的location处理， 请求被分派到监听在localhost:9000的FastCGI服务器。fastcgi_param 指令设置FastCGI 参数 SCRIPT_FILENAME 为 “/data/www/index.php”, 然后FastCGI 服务器执行这个文件. 变量 $document_root 等同于root指令的值，而变量 $fastcgi_script_name 等同于请求 URI, 例如 “/index.php”.
- 请求 “/about.html” 仅仅匹配前缀location“/”, 因此, 它在这个location中被处理. 通过指令 “root /data/www” 请求被映射为文件/data/www/about.html, 然后这个文件被发送到客户端。
- 处理请求 “/”要更复杂一些. 它仅仅匹配前缀location“/”, 因此, 它在这个location中被处理. 然后index指令根据它的参数和“root /data/www”指令来检验index文件的存在。如果文件/data/www/index.html不存在，而文件 /data/www/index.php 存在, 则指令执行一个内部重定向到“/index.php”, 然后 nginx 重新搜索location就像这个请求是从客户端发过来一样。如我们前面所见，这个重定向请求最终将被FastCGI服务器处理。

服务器名称

> 注：内容翻译自Nginx官网文档 Server Name。

服务器名称通过使用server_name指令来定义并决定哪个服务器块(server block)将用于给定的请求。参考"How nginx processes a request"。

> 注： 中文翻译版本 nginx如何处理请求。

可以使用精确名称，通配符和正则表达式：

```
server {
    listen       80;
    server_name  example.org  www.example.org;
    ...
}

server {
    listen       80;
    server_name  *.example.org;
    ...
}

server {
    listen       80;
    server_name  mail.*;
    ...
}

server {
    listen       80;
    server_name  ~^(?<user>.+)\.example\.net$;
    ...
}
```

当通过名称搜索虚拟服务器时， 如果名字和多个指定的变量匹配， 例如同时匹配通配符和正则表达式，在下面的优先级次序中，第一个匹配的变量将被选择：

1. 精确名称
2. 星号开头的最长的通配符名称, 例如 "*.example.org"
3. 星号结束的最长的通配符名称, 例如 "mail.*"
4. 第一个匹配的正则表达式(按照出现在配置文件中的顺序)

通配符名称

通配符名称可以在名称的开头和结尾包含星号，并且只能紧挨着点号(.)。名称"www.*.example.org"*和*"w.*example.org"是不合法的。当然，这些名字可以用正则表达式来指定，例如："~^www..+.example.org$" 和 "~^w.*.example.org$".* 星号可以匹配多个名称部位，名称 “.example.org” 不仅可以匹配 www.example.org 还可以匹配 www.sub.example.org.

".example.org"这种特殊的通配符名称可以用于匹配精确名称"example.org"和通配符名称"*.example.org".

正则表达式名称

nginx所用的正则表达式兼容于Perl编程语言(PCRE)。为了使用正则表达式， 服务器名称必须以波浪号(~)开头：

```
server_name  ~^www\d+\.example\.net$;
```

否则将被当成是精准名称，或者如果表达式中包含星号就被当成通配符名称(而且大都被认为时不合法).不要忘记设置"^"和"$"锚点。虽然语法上没要求，但是逻辑上需要他们。还要注意域名的点号要使用反斜杠做转义。包含字符"{"和"}"的正则表达式需要使用引号：

```
server_name  "~^(?<name>\w\d{1,3}+)\.example\.net$";
```

否则nginx会启动失败并显示错误信息：

```
directive "server_name" is not terminated by ";" in ...
```

被命名的正则表达式捕获器可以随后作为变量使用：

```
server {
    server_name   ~^(www\.)?(?<domain>.+)$;

    location / {
        root   /sites/$domain;
    }
}
```

PCRE 类库使用下列语法来支持命名捕获器：

```
?<name>            Perl 5.10 兼容语法, 从PCRE-7.0开始支持
?'name'            Perl 5.10 兼容语法, 从PCRE-7.0开始支持
?P<name>        Python 5.10 兼容语法, 从PCRE-4.0开始支持
```

五花八门的名称

有一些服务器名称需要特别对待。

如果一个非"default"的服务器块需要处理不带"Host" header的请求， 需要指定一个空的名称：

```
server {
    listen       80;
    server_name  example.org  www.example.org  "";
    ...
}
```

服务器块中如果没有定义server_name，那么nginx将使用空名称作为服务器名。

直到0.8.48版本，nginx在这种情况下使用机器的hostname作为服务器名称。如果服务器名称被定义为"$hostname"(0.9.4), 使用机器的hostname。

如果某些请求使用IP地址替代服务器名称， 请求的"Host" header将包含IP地址， 使用IP地址作为服务器名称可以处理这些请求：

```
server {
    listen       80;
    server_name  example.org
                 www.example.org
                 ""
                 192.168.1.1
                 ;
    ...
}
```

在这个匹配所有的服务器例子中， 可以看到一个奇怪的名称"_":

```
server {
    listen       80  default_server;
    server_name  _;
    return       444;
}
```

这个名字没有任何特别之处，它仅仅是无数从来不和实际名称相交的非法域名中的一个。其他非法名称类似"--" 和 "!@#"。

0.6.25版本之前的nginx支持特殊的名称""，被错误的理解为是一个匹配所有的名称，但是实际上从来没有工作过。相反，这个功能现在是通过*server_name_in_redirect*指令来提供的。特殊名称""现在被废弃，应该使用server_name_in_redirect指令。注意使用server_name指令时

是没有方法可以指定匹配所有的名称或者默认服务器的。这是listen指令的一个属性，而不是server_name指令。请参考How nginx processes a request。可以定义服务器监听于端口*:80*和:8080，并指示某个成为端口*:8080*的默认服务器，而其他成为端口:80的默认服务器：

```
server {
    listen       80;
    listen       8080  default_server;
    server_name  example.net;
    ...
}

server {
    listen       80  default_server;
    listen       8080;
    server_name  example.org;
    ...
}
```

优化

精确名称，以星号开头的通配符名称和以星号结束的通配符名称存储于绑定在监听端口上的三个hash表中。hash表的大小在配置阶段做了优化以便可以以最大的CPU缓存命中来查找服务器。构建hash table的细节在单独的文档中提供。

精确名称的hash表被第一个搜索。如果名称没有找到，继续搜索用星号开头的通配符名称的hash表。如果名字还没有找到，继续搜索用星号结束的通配符名称的hash表。

搜索通配符名称哈希表比搜索精确名称哈希表要慢，因为名称是通过域名部分来搜索的。注意特殊形式的通配符".example.org"是存储在通配符名称哈希表而不是精确名称哈希表。

正则表达式是逐个检验的，因此是最慢的方法，而且不可扩展。

处于这些理由， 最好能尽可能的使用精确名称。例如， 如果一个服务器最频繁的请求名称是example.org和www.example.org， 显式定义他们将更有效率：

```
server {
    listen       80;
    server_name  example.org  www.example.org  *.example.org;
    ...
}
than to use the simplified form:

server {
    listen       80;
    server_name  .example.org;
    ...
}
```

如果定义有大量的服务器名称，或者定义有非正常的长服务器名称， 有必要在http级别调整 server_names_hash_max_size 和 server_names_hash_bucket_size 指令。 server_names_hash_bucket_size指令的默认值可能是32,或者64,或者其他值，取决于CPU cache line的大小。如果默认值是32而服务器名被定义为"too.long.server.name.example.org", 那么nginx在启动时会失败并显示错误信息：

```
could not build the server_names_hash,
you should increase server_names_hash_bucket_size: 32
```

在这种情况下， 指令值应该增加到下一个级别：

```
http {
    server_names_hash_bucket_size  64;
    ...
```

如果定义有大量的服务器名称，会出现另外一个错误：

```
could not build the server_names_hash,
you should increase either server_names_hash_max_size: 512
or server_names_hash_bucket_size: 32
```

在这种情况下，先尝试将server_names_hash_max_size设置为接近服务器名称的数量。只有在这种方式无效，或者nginx的启动时间长的不可接收时，尝试增加 server_names_hash_bucket_size。

如果某台服务器是该监听接口唯一的服务器，则nginx将完全不检验服务器名字(也不会为监听端口构建哈希表)。但是，有一个例外。如果服务器名称是带有捕获器的正则表达式， 那么nginx将不得不执行表达式以便获取捕获的内容。

兼容性

- 特殊服务器名称"$hostname"从0.9.4版本开始支持
- 在0.8.48版本之后，默认服务器名称是空名称""
- 被命名的正则表达式服务器名称捕获器从0.8.25版本开始支持
- 正则表达式服务器名称捕获器从0.7.40版本开始支持
- 空服务器名称""从0.7.12版本开始支持
- 从0.6.25版本开始支持使用通配符服务器名称或者正则表达式名称作为第一个服务器名称
- 从0.6.7版本开始支持使用正则表达式名称
- 从0.6.0版本开始支持"example.*"形式的通配符
- 从0.3.18版本开始支持特别格式.example.org
- 从0.1.13版本开始支持通配符形式*.example.org

使用nginx实现HTTP负载均衡

> 注：内容翻译自Nginx官网文档 Using nginx as HTTP load balancer。

介绍

在多个应用实例间做负载均衡是一个被广泛使用的技术，用于优化资源效率，最大化吞吐量，减少延迟和容错。

nginx可以作为一个非常高效的HTTP负载均衡器来分发请求到多个应用服务器，并提高web应用的性能，可扩展性和可靠性。

负载均衡方法

nginx支持以下负载均衡机制（或者方法）：

- round-robin/轮询： 到应用服务器的请求以round-robin/轮询的方式被分发
- least-connected/最少连接：下一个请求将被分派到活动连接数量最少的服务器
- ip-hash/IP散列： 使用hash算法来决定下一个请求要选择哪个服务器(基于客户端IP地址)

默认负载均衡配置(轮询)

nginx中最简单的负载均衡配置看上去大体如下：

```
http {
    upstream myapp1 {
        server srv1.example.com;
        server srv2.example.com;
        server srv3.example.com;
    }

    server {
        listen 80;

        location / {
            proxy_pass http://myapp1;
        }
    }
}
```

在上面的例子中， 同一个应用有3个实例分别运行在srv1-srv3。当没有特别指定负载均衡方法时， 默认为round-robin/轮询。所有请求被代理到服务器集群myapp1， 然后nginx实现HTTP负载均衡来分发请求。

在nginx中反向代理的实现包括HTTP, HTTPS, FastCGI, uwsgi, SCGI, 和 memcached的负载均衡。

要配置负载均衡用HTTPS替代HTTP，只要使用"https"作为协议即可。

为FastCGI, uwsgi, SCGI, 或 memcached 搭建负载均衡时， 只要使用相应的fastcgi_pass, uwsgi_pass, scgi_pass, 和 memcached_pass指令。

最少连接负载均衡

另一个负载均衡方式是least-connected/最少连接。当某些请求需要更长时间来完成时，最少连接可以更公平的控制应用实例上的负载。

使用最少连接负载均衡时，nginx试图尽量不给已经很忙的应用服务器增加过度的请求， 而是分配新请求到不是那么忙的服务器实例。

nginx中通过在服务器集群配置中使用least_conn指令来激活最少连接负载均衡方法：

```
upstream myapp1 {
    least_conn;
    server srv1.example.com;
    server srv2.example.com;
    server srv3.example.com;
}
```

会话持久化(ip-hash)

请注意，在轮询和最少连接负载均衡方法中，每个客户端的后续请求被分派到不同的服务器。对于同一个客户端没有任何方式保证发送给同一个服务器。

如果需要将一个客户端绑定给某个特定的应用服务器——用另一句话说，将客户端会话"沾住"或者"持久化"，以便总是能选择特定服务器——，那么可以使用ip-hash负载均衡机制。

使用ip-hash时，客户端IP地址作为hash key使用，用来决策选择服务器集群中的哪个服务器来处理这个客户端的请求。这个方法保证从同一个客户端发起的请求总是定向到同一台服务器，除非服务器不可用。

要配置使用ip-hash负载均衡，只要在服务器集群配置中使用ip_hash指令：

```
upstream myapp1 {
    ip_hash;
    server srv1.example.com;
    server srv2.example.com;
    server srv3.example.com;
}
```

带权重的负载均衡

可以通过使用服务器权重来影响nginx的负载均衡算法。

在上面的例子中，服务器权重没有配置，这意味着所有列出的服务器被认为对于具体的负载均衡方法是完全平等的。

特别是轮询，分派给服务器的请求被认为是大体相当的——假设有足够的请求，并且这些请求被以同样的方式处理而且完成的足够快。

当服务器被指定weight/权重参数时，负载均衡决策会考虑权重。

```
upstream myapp1 {
    server srv1.example.com weight=3;
    server srv2.example.com;
    server srv3.example.com;
}
```

With this configuration, every 5 new requests will be distributed across the application instances as the following: 3 requests will be directed to srv1, one request will go to srv2, and another one — to srv3.

在这个配置中，每5个新请求将会如下的在应用实例中分派： 3个请求分派去srv1,一个去srv2,另外一个去srv3.

在最近的nginx版本中，可以类似的在最少连接和IP哈希负载均衡中使用权重。

健康检查

nginx中的反向代理实现包含in-band/带内(或者说被动)的服务器健康检查。如果某台服务器响应失败，nginx将标记这台服务器为"失败"，之后的一段时间将尽量避免选择这台服务器来处理后续请求。

max_fails 指令设置在fail_timeout时间内和服务器通讯连续不成功尝试的数量。默认，max_fails设置为0.如果设置为0, 则关闭这台服务器的健康检查。fail_timeout参数同样也定义了服务器被标记为"失败"的时间长度。

在服务器失败之后的fail_timeout间隔时间后， nginx会开始温和的用来自实际客户端的请求来探测服务器。如果探测成功， 服务器将被标记是存活。

更多

此外，在nginx中还有更多的指令和参数可以控制服务器负载均衡。例如：proxy_next_upstream, backup, down, 和 keepalive。请查阅我们的参考文档来获取更多信息。

> 注： 后面有翻译的 HTTP Upstream模块 文档。

最后， 应用负载均衡，应用健康检查， 活动监控和on-the-fly 服务器集群重配置在付费的nginx plus中提供。

HTTP Upstream模块(翻译)

> 注：内容翻译自Nginx官网文档 HTTP Upstream模块，个别地方稍作补充。

ngx_http_upstream_module 模块用于定义可以被proxy_pass, fastcgi_pass, uwsgi_pass, scgi_pass和memcached_pass指令引用的服务器集群。

配置范例

```
upstream backend {
    server backend1.example.com       weight=5;
    server backend2.example.com:8080;
    server unix:/tmp/backend3;

    server backup1.example.com:8080   backup;
    server backup2.example.com:8080   backup;
}

server {
    location / {
        proxy_pass http://backend;
    }
}
```

动态可配置集群(Dynamically configurable group)是商业版本(commercial subscription)的一部分.

```
resolver 10.0.0.1;

upstream dynamic {
    zone upstream_dynamic 64k;

    server backend1.example.com      weight=5;
    server backend2.example.com:8080 fail_timeout=5s slow_start=30s;
    server 192.0.2.1                 max_fails=3;
    server backend3.example.com      resolve;

    server backup1.example.com:8080  backup;
    server backup2.example.com:8080  backup;
}

server {
    location / {
        proxy_pass http://dynamic;
        health_check;
    }
}
```

指令

upstream指令

upstream指令用于定义服务器集群。服务器可以监听在不同端口。另外，监听在TCP和UNIX-domain socket的服务器可以混合使用

```
Syntax:    upstream name { ... }
Default:    —
Context:    http
```

范例：

```
upstream backend {
    server backend1.example.com weight=5;
    server 127.0.0.1:8080       max_fails=3 fail_timeout=30s;
    server unix:/tmp/backend3;

    server backup1.example.com  backup;
}
```

默认，使用带权重的round-robin平衡算法将请求分派到服务器。在上面的例子中， 每7个请求将被如下分配：

- 5个请求去backend1.example.com
- 1个请求去127.0.0.1:8080
- 1个请求去unix:/tmp/backend3

在和某台服务器通讯的过程中，如果发生错误， 请求将被分派给下一个服务器， 以此类推知道所有可用服务器都被尝试。如果没有任何一个服务器可以可以返回成功的应答，则客户端将会收到和最后一台机器的通讯结果。

server指令

server指令用于定义一台服务器的地址和其他参数。地址可以是域名或者IP地址，端口可选，或者是以"unix:"前缀指定的UNIX-domain socket路径。如果端口没有指定，将使用80端口。可以解析为多个IP地址的域名将一次性定义多台服务器。

```
Syntax:    server address [parameters];
Default:    —
Context:    upstream
```

下面是可用的参数列表。

- weight=number

  设置服务器的权重，默认为1.

- max_fails=number

  设置在参数fail_timeout指定的时间内，发生的和服务器通讯的不成功的数量。默认情况，不成功尝试次数被设置为1.如果设置为0则关闭尝试计数。至于什么是不成功的尝试则通过proxy_next_upstream, fastcgi_next_upstream, uwsgi_next_upstream, scgi_next_upstream, 和 memcached_next_upstream指令定义。

- fail_timeout=time

  用于设置：

  - 时间段，在此期间应该发起指定数量的和服务器通讯的不成功尝试，以判断服务器是否不可到达
  - 和接下来服务器被判定为不可到达的时间期间

  默认，fail_timeout参数被设置为10秒钟.

  > 注：如果设置为max_fails=5;fail_timeout=30s，表示如果有5次请求失败，则该服务器被断定为不可到达，之后30s之类将不再尝试这台机器。再之后的每30s，都将进行最多5次尝试，如果继续失败则继续判断为不可到达并不再尝试。

- backup

  标记当前服务器为备用服务器。当主服务器(注：应该是没有标记为backup和down的服务器)都不能达到时请求将被分派过去。

- down

  将当前服务器标记为永久不可到达。

另外，商业版本将支持下面的参数：

- max_conns=number

  限制同时激活的到被代理的服务器的最大连接数。默认值为0,表示没有限制。

  当长连接**(keepalive connection)**和多**work**被启用时， 到被代理的服务器的最大连接总数可能超过**max_conns**参数的值

- resolve

  监视域名对应的服务器IP地址的变化，并自动修改upstream配置而不需要重启nginx服务器(1.5.12版本)。The server group must reside in the shared memory

  > 注：这句不懂....

  为了让这个参数工作，resolver必须指定在http块中。例如：

  ```
  http {
      resolver 10.0.0.1;

      upstream u {
          zone ...;
          ...
          server example.com resolve;
      }
  }
  ```

- route=string

  设置服务器路由名字(route name)

- slow_start=time

  设置时间段，在此期间服务器将从0到正常值逐渐恢复它的权重，当服务器从不健康变成健康，或者服务器从被标记为不可到达一段时间后变成可到达时。默认值是0, 表示关闭缓慢启动功能。

如果服务器集群中仅有一台服务器， **max_fails, fail_timeout** 和 **slow_start** 参数都将被忽略， 而这台机器永远不会被标记为不可到达。

zone指令(商业版)

zone指令在版本1.9.0中出现。

```
Syntax:    zone name [size];
Default:    —
Context:    upstream
```

定义共享内存zone的名称和大小， 共享内存zone用于保存集群配置和运行时状态，以便在worker进程之间共享。多个集群可以分享同一个zone。在这种情况下，只需要指定一次大小。

另外，作为商业版本的一部分，这样的集群容许改变集群成员或者修改某台服务器的设置而不需要重启nginx。

state指令(商业版)

state指令在版本1.9.7中出现，属于商业版本的一部分.

```
Syntax:    state file;
Default:    —
Context:    upstream
```

指定一个文件来保存动态可配置集群的状态。状态目前仅限于服务器列表及其参数.当解析配置时state文件被读取并在每次upstream配置被修改时更新。直接修改这个文件的内容是无效的。这个指令不能server指令一起使用。

configuration reload 或者 binary upgrade造成的修改可能丢失。

hash指令

hash指令在版本1.7.2中出现。

```
Syntax:    hash key [consistent];
Default:    —
Context:    upstream
```

指定服务器集群的负载均衡方法，客户端-服务器映射基于散列key值。key可以包含文本，变量和他们的组合。注意从集群中添加或者删除一个服务器可能导致大部分key重新映射到不同的服务器。这个方法兼容Cache::Memcached Perl类库。

如果consistent参数被指定，则将使用ketama一致性哈希算法。这个算法确保当有一台服务器添加到集群或者从集群中删除时仅有少数key被重新映射到不同的服务器。这将有助于缓存系统实现更高的缓存命中率。ketama_points参数设置为160时，这个方法兼容 Cache::Memcached::Fast Perl 类库。

ip_hash指令

```
Syntax:    ip_hash;
Default:    —
Context:    upstream
```

指定集群使用的负载均衡算法，基于客户端IP地址将请求分派给服务器。客户端IPv4地址的前三个 八位字节，或者整个IPv6地址被作为哈希key来使用。这个算法保证从同一个客户端来的请求总是被分派到同样的服务器，除非这个服务器不可达到。后面这种情况下客户端请求将被分派到其他服务器。大多数情况，请求总是被分派到同一个服务器。

IPv6地址的支持是从版本1.3.2 和 1.2.2开始。

如果某台服务器需要被永久移除，那么应该将它标记为down以便保持当前的客户端IP地址的哈希。

例如:

```
upstream backend {
    ip_hash;

    server backend1.example.com;
    server backend2.example.com;
    server backend3.example.com down;
    server backend4.example.com;
}
```

在版本1.3.1 和 1.2.2之前，无法指定使用ip_hash负载均衡算法的服务器的权重。

> 注：言下之意，这两个版本之后就可以指定权重了？）

keepalive指令

keepalive指令出现在版本1.1.4。

```
Syntax:    keepalive connections;
Default:    —
Context:    upstream
```

激活到upstream服务器的连接缓存（注：即长连接）。

connections参数设置每个worker进程在缓冲中保持的到upstream服务器的空闲**keepalive**连接的最大数量.当这个数量被突破时，最近使用最少的连接将被关闭。

特别提醒：keepalive指令不会限制一个nginx worker进程到upstream服务器连接的总数量。connections参数应该设置为一个足够小的数字来让upstream服务器来处理新进来的连接。

> 注： 这句话的语义非常的费解，原文如下：
>
> The connections parameter should be set to a number small enough to let upstream servers process new incoming connections as well. 我的理解是如果想让upstream每次都处理新的进来的连接，就应该将这个值放的足够小。反过来理解，就是如果不想让upstream服务器处理新连接，就应该放大一些？

使用keepalive连接的memcached upstream配置的例子：

```
upstream memcached_backend {
    server 127.0.0.1:11211;
    server 10.0.0.2:11211;

    keepalive 32;
}

server {
    ...

    location /memcached/ {
        set $memcached_key $uri;
        memcached_pass memcached_backend;
    }
}
```

对于HTTP，proxy_http_version指定应该设置为"1.1"，而"Connection" header应该被清理：

```
upstream http_backend {
    server 127.0.0.1:8080;

    keepalive 16;
}

server {
    ...

    location /http/ {
        proxy_pass http://http_backend;
        proxy_http_version 1.1;
        proxy_set_header Connection "";
        ...
    }
}
```

或者，HTTP/1.0 持久连接可以通过传递"Connection: Keep-Alive" header 到upstream server， 但是不推荐使用这种方法。

对于FastCGI服务器，要求设置fastcgi_keep_conn来让长连接工作：

```
upstream fastcgi_backend {
    server 127.0.0.1:9000;

    keepalive 8;
}

server {
    ...

    location /fastcgi/ {
        fastcgi_pass fastcgi_backend;
        fastcgi_keep_conn on;
        ...
    }
}
```

当使用默认的round-robin之外的负载均衡算法时，必须在keepalive指令之前激活他们。

SCGI 和 **uwsgi** 协议没有**keepalive**连接的概念。

ntlm指令(商业版)

ntlm指令出现在版本 1.9.2 中。

```
Syntax:    ntlm;
Default:    —
Context:    upstream
```

容许使用NTLM算法代理请求。一旦客户端发送一个带有"Authorization" header 并且值是"Negotiate" 或 "NTLM"开头时，upstream 连接被绑定到这个客户端连接。后续的客户端请求将被通过同样一个upstream连接代理，以保持认证的上下文。

为了让NTLM算法工作，必须开启到upstream服务器的keepalive连接。proxy_http_version指定应该设置为"1.1"，而"Connection" header应该被清理：

```
upstream http_backend {
    server 127.0.0.1:8080;

    ntlm;
}

server {
    ...

    location /http/ {
        proxy_pass http://http_backend;
        proxy_http_version 1.1;
        proxy_set_header Connection "";
        ...
    }
}
```

当使用默认的round-robin之外的负载均衡算法时，必须在ntlm指令之前激活他们。

ntlm指令是商业版本的一部分。

least_conn指令

least_conn指令出现于版本 1.3.1 和 1.2.2.

```
Syntax:    least_conn;
Default:    —
Context:    upstream
```

指定集群应该使用的负载均衡方法，分派请求到活动连接数量最少的服务器， 兼顾服务器权重。如果有多台这样的服务器，这些服务器将尝试轮流使用带权重的round-robin平衡算法。

> 这句话不太理解，原文： If there are several such servers, they are tried in turn using a weighted round-robin balancing method. 我的理解，如果有多台服务器的连接数都相同，都是最小，这是这几台服务器之间选择的算法就用带权重的round-robin？

least_time指令(商业版)

least_time指令出现于把版本1.7.10.

```
Syntax:    least_time header | last_byte;
Default:    —
Context:    upstream
```

指定集群应该使用的负载均衡方法，分派请求到平均响应时间最快和活动连接数量最少的服务器， 兼顾服务器权重。如果有多台这样的服务器，这些服务器将尝试轮流使用带权重的round-robin平衡算法。

如果header参数被指定，使用接收到响应header的时间;如果last_byte参数被指定，使用接收到完整响应的时间。

least_time指令是商业版本的一部分。

health_check指令(商业版)

开启对所在location引用的集群中的服务器的定期健康检查。

```
Syntax:    health_check [parameters];
Default:    —
Context:    location
```

下面可选参数将被支持：

- interval=time

  设置两次连续健康检查之间的间隔时间，默认5秒钟。

- fails=number

  设置连续失败的健康检查次数，之后这台服务器会被认为是不健康，默认为1.

- passes=number

  设置连续通过的健康检查次数，之后这台服务器会被认为是健康的，默认为1.

- uri=uri

定义健康检查请求的URL，默认是"/".

- match=name

  指定匹配块来配置测试可以通过健康检查的响应。默认，响应的状态码应该是2xx 或者 3xx;

- port=number

  定义到执行健康检查的服务器的连接端口(1.9.7)。默认，和服务器端口相同。

例如：

```
location / {
    proxy_pass http://backend;
    health_check;
}
```

将每5秒钟发送"/"请求给后端集群中的每台服务器。如果发生任何通讯错误或者超时，或者被代理的服务器返回的状态码不是2xx 或者 3xx，则健康检查失败，服务器将被认为是不健康。客户端请求不会发送给不健康的服务器。

健康检查可以配置为检验响应的状态码，是否存在特定header和他们的值，还有http body内容。检验可以使用match指令单独配置并在match参数中引用。例如：

```
http {
    server {
    ...
        location / {
            proxy_pass http://backend;
            health_check match=welcome;
        }
    }

    match welcome {
        status 200;
        header Content-Type = text/html;
        body ~ "Welcome to nginx!";
    }
}
```

这个配置表明，为了通过健康检查，健康检查请求的应答应该成功，状态是200, content type 是 "text/html"，并在body中包含"Welcome to nginx!".

The server group must reside in the shared memory.

如果同一个集群的服务器定义有多个健康检查，任何一个检查失败都会导致对应的服务器被认为是不健康。

请注意：当被用于健康检查时，大部分变量都将是空值。

health_check指令是商业版本的一部分。

match指令(商业版)

```
Syntax:    match name { ... }
Default:    —
Context:    http
```

定义命名的检验集合用于验证健康检查请求的应答。

下面的选项可以在应答中检验：

- status 200;

  状态码是 200

- status ! 500;

  状态码不是 500

- status 200 204;

  状态码是 200 或 204

- status ! 301 302;

  状态码是 301 或 302

- status 200-399;

  状态码在 200 到 399 的范围内

- status ! 400-599;

  状态码不在 400 到 599 的范围内

- status 301-303 307;

  状态码是 301, 302, 303, 或 307

- header Content-Type = text/html;

  header包含“Content-Type” 并且值是 text/html

- header Content-Type != text/html;

  header包含“Content-Type” 并且值不是 text/html

- header Connection ~ close;

  header包含“Connection” 并且值匹配正则表达式 close

- header Connection !~ close;

  header包含“Connection” 并且值不匹配正则表达式 close

- header Host;

  header包含“Host”

- header ! X-Accel-Redirect;

  header不带 “X-Accel-Redirect”

- body ~ "Welcome to nginx!";

  body 匹配正则表达式 “Welcome to nginx!”

- body !~ "Welcome to nginx!";

  body 不匹配正则表达式 “Welcome to nginx!”

如果指定有多个检验，响应必须配置所有检验。

仅检查响应**body**的前**256K**

例如:

```
# status is 200, content type is "text/html",
# and body contains "Welcome to nginx!"
match welcome {
    status 200;
    header Content-Type = text/html;
    body ~ "Welcome to nginx!";
}
# status is not one of 301, 302, 303, or 307, and header does not have "Refresh:"
match not_redirect {
    status ! 301-303 307;
    header ! Refresh;
}
# status ok and not in maintenance mode
match server_ok {
    status 200-399;
    body !~ "maintenance mode";
}
```

match指令是商业版本的一部分。

queue指令(商业版)

queue指令出现于版本1.5.12.

```
Syntax:    queue number [timeout=time];
Default:    —
Context:    upstream
```

当处理请求时，如果某台upstream服务器无法立即被选择，并且集群中的其他服务器都已经达到了max_conns限制，请求将被放置在queue中。如果queue满了，或者要分派请求的服务器无法在timeout参数所设置的时间内选择请求，将会给客户端返回502(Bad Gateway)错误。

timeout参数的默认值是60秒。

queue指令是商业版本的一部分。

sticky指令(商业版)

sticky指令出现于版本1.5.7.

```
Syntax:
    sticky cookie name [expires=time] [domain=domain] [httponly] [secure] [path=path];
    sticky route $variable ...;
    sticky learn create=$variable lookup=$variable zone=name:size [timeout=time];
Default:    —
Context:    upstream
```

开启session affinity(注：翻译为会话保持？)特性， 从同样的客户端发出的请求被分派到服务器集群中同样的服务器上。有三个方法可选：

cookie

当使用cookie方法时， 被指派的服务器信息将在nginx生成的HTTP cookie中传递：

```
upstream backend {
    server backend1.example.com;
    server backend2.example.com;

    sticky cookie srv_id expires=1h domain=.example.com path=/;
}
```

如果客户端还没有被绑定到特定的服务器，请求进来时将被分派到配置的均衡算法选择的服务器。后续带有cookie的请求将被分派给指派的服务器。如果指派的服务器无法处理请求，新的服务器将被选择，就如同客户端没有被绑定一样。

第一个参数设置cookie的名字。其他参数如下：

- expires=time

  设置浏览器保存cookie的时间。特殊值"max"将设置cookie过期时间为"31 Dec 2037 23:55:55 GMT".如果这个参数没有指定， cookie将在浏览器session结束时过期。

- domain=domain

  定义cookie设置的domain

- httponly

  添加HttpOnly属性到cookie(1.7.11)

- secure

  添加Secure属性到cookie(1.7.11)

- path=path

  定义cookie设置的path属性。

以上任何一个参数，如果缺失则对应的**cookie**就不会被设置

route

当使用route方法时，代理服务器基于第一次请求的接收情况给客户端分派一个路由(route)。这个客户端随后的所有请求将在cookie或者URI中携带路由信息。这些信息类似于server指令中的route参数。如果被指派的服务器无法处理请求，新的服务器将被选择，就如同请求中没有route信息一样

route方法的参数知名可能包含路由信息的变量。第一个非空变量将用于查找匹配的服务器。

例如:

```
map $cookie_jsessionid $route_cookie {
    ~.+\.(?P<route>\w+)$ $route;
}

map $request_uri $route_uri {
    ~jsessionid=.+\.(?P<route>\w+)$ $route;
}

upstream backend {
    server backend1.example.com route=a;
    server backend2.example.com route=b;

    sticky route $route_cookie $route_uri;
}
```

这里，如果请求中存在"JSESSIONID" cookie，则从"JSESSIONID" cookie中获取到路由信息。否在，从URI中取。

learn

When the learn method (1.7.1) is used, nginx analyzes upstream server responses and learns server-initiated sessions usually passed in an HTTP cookie.

当使用learn方法时，nginx分析upstream服务器的响应并学习通常在HTTP cookie中传递的 server-initiated 会话。

```
upstream backend {
   server backend1.example.com:8080;
   server backend2.example.com:8081;

   sticky learn
          create=$upstream_cookie_examplecookie
          lookup=$cookie_examplecookie
          zone=client_sessions:1m;
}
```

在这个例子中， upstream server通过在应答中设置cookie "EXAMPLECOOKIE"创建会话。带有这个cookie的后续请求将被分派到同一个服务器。如果服务器不能处理请求，新的服务器将被选择，就如同客户端没有被绑定一样。

参数create 和 lookup 分别指定变量来指示如何创建新会话和搜索已经存在的会话。两个参数都可以指定多个，这样第一个非空的变量将被使用。

会话存储在shared memory zone，名字和大小在zone属性中配置。在64位平台上一个 megabyte zone可以存储大概8000个会话。在timeout参数指定的期间内没有被访问的会话将被从zone上移除。默认，超时时间设置为10分钟。

sticky指令是商业版本的一部分。

sticky_cookie_insert指令(废弃)

```
Syntax:    sticky_cookie_insert name [expires=time] [domain=domain] [path=path];
Default:    —
Context:    upstream
```

sticky_cookie_insert指令在版本1.5.7之后就被废弃了。被sticky指令替代。